

NumPy

November 30, 2024

1 Learn NumPy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

sequence size and speed are particularly important in scientific computing

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called axes. For example, the array for the coordinates of a point in 3D space, `[1, 2, 1]`, has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

1.0.1 Difference between Numpy and standard Python

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

```
[305]: [[1., 0., 0.],  
        [0., 1., 2.]]
```

```
[305]: [[1.0, 0.0, 0.0], [0.0, 1.0, 2.0]]
```

NumPy's array class is called `ndarray`. It is also known by the alias `array`. Note that `numpy.array` is not the same as the Standard Python Library class `array.array`, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an `ndarray` object are 1. `ndarray.n`: the number of axes (dimensions) of the array

2. `ndarray.shape`: the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For matrix with `n` rows and `m` columns, `shape` will be `(n,m)`. The length of the tuple is the number of axes, `nd`
3. `ndarray.size`: the total number of elements of the array. This is equal to the product of the elements of `shape`
4. `ndarray.dtype`: an object describing the type of the elements in the array. One can create or specify `dtype`'s using standard Python types. Additionally NumPy provides types of its own. `numpy.int32`, `numpy.int16`, and `numpy.float64` are some
5. `ndarray.itemsize`: the size in bytes of each element of the array. For example, an array of elements of type `float64` has `itemsize 8` ($=64/8$), while one of type `complex32` has `itemsize 4` ($=32/8$). It is equivalent to `ndarray.dtype.itemsize`
6. `ndarray.data`: the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

```
[306]: import numpy as np  
np.__version__
```

```
[306]: '1.26.4'
```

```
[307]: list1 = [1,2,3,4,5]  
list1
```

```
[307]: [1, 2, 3, 4, 5]
```

1.0.2 Basic Attributes of the ndarray Class

```
[308]: list2 = ['mukesh', 28, 10000000, [1,2,3,4,5,6], True]
list2
```

```
[308]: ['mukesh', 28, 10000000, [1, 2, 3, 4, 5, 6], True]
```

This doesnot work with massive amount of data as it will become very slow. so NumPy was introduces wich is every fast but allows to manipulate similar kind of data.

```
[309]: import numpy as np
```

```
[310]: #help(np.ndarray)
```

Attribute	Description
Shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
Size	The total number elements in the array.
Ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

```
[311]: np1 = np.array([0,1,2,3,4,5,6,7,8,9]) #creatin NumPy array
np1 #display the values of NumPy array
```

```
[311]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[312]: type(np1)
```

```
[312]: numpy.ndarray
```

```
[313]: np1.shape
```

```
[313]: (10,)
```

```
[314]: np1.size
```

```
[314]: 10
```

```
[315]: np1.ndim
```

```
[315]: 1
```

```
[316]: np1.nbytes
```

```
[316]: 80
```

```
[317]: np1.dtype
```

```
[317]: dtype('int64')
```

1.0.3 Data Types

dtype	Variants	Description
int	int8, int16, int32, int64	Integers
uint	uint8, uint16, uint32, uint64	Unsigned (nonnegative) integers
bool	Bool	Boolean (True or False)
float	float16, float32, float64, float128	Floating-point numbers
complex	complex64, complex128, complex256	Complex-valued floating-point numbers

1.0.4 3 Ways to Creating NumPy Arrays

1st method: via python list

```
[318]: mylist = [1,2,3,4,5]
mylist_from_array = np.array(mylist)
mylist_from_array
```

```
[318]: array([1, 2, 3, 4, 5])
```

```
[319]: mylist_from_array * 10
```

```
[319]: array([10, 20, 30, 40, 50])
```

2nd Method: via Python Tuples

```
[320]: # creation via Python Tuple
mytuple = (1,2,3,4,5,6,7,8,9,10)
np.array(mytuple)
```

```
[320]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

3rd method: via NumPy arange method

```
[321]: np.arange(10)
```

```
[321]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[322]: np.arange(10, 23)
```

```
[322]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22])
```

```
[323]: np.arange(10,23,5)
```

```
[323]: array([10, 15, 20])
```

The following example demonstrates how to use the dtype attribute to generate arrays of integer-, float-, and complex-valued elements:

```
[324]: npp = np.array([1,2,3,4], dtype=np.int16)
print(npp)
```

```
[1 2 3 4]
```

```
[325]: npp1 = np.array([1,2,3,4,5,], dtype=np.float32)
print(npp1)
```

```
[1. 2. 3. 4. 5.]
```

```
[326]: npp2 = np.array([1,2,3,4,5,6], dtype=np.complex64)
npp2
```

```
[326]: array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j, 6.+0.j], dtype=complex64)
```

Once a NumPy array is created, its dtype cannot be changed

```
[327]: data = np.array([1, 2, 3], dtype=np.float32)
data
```

```
[327]: array([1., 2., 3.], dtype=float32)
```

```
[328]: data.dtype
```

```
[328]: dtype('float32')
```

```
[329]: data = np.array(data, dtype=np.int32)
data.dtype
```

```
[329]: dtype('int32')
```

```
[330]: data
```

```
[330]: array([1, 2, 3], dtype=int32)
```

It can also be changed using the astype method of the ndarray class

```
[331]: data = np.array([1, 2, 3], dtype=np.float32)
data
```

```
[331]: array([1., 2., 3.], dtype=float32)
```

```
[332]: data.astype(np.int32)
```

```
[332]: array([1, 2, 3], dtype=int32)
```

```
[333]: d1 = np.array([1, 2, 3], dtype=float)
d2 = np.array([1, 2, 3], dtype=complex)
d1 + d2
```

```
[333]: array([2.+0.j, 4.+0.j, 6.+0.j])
```

```
[334]: (d1 + d2).dtype
```

```
[334]: dtype('complex128')
```

Real and Imaginary Parts

```
[335]: data = np.array([1, 2, 3], dtype=complex)
data
```

```
[335]: array([1.+0.j, 2.+0.j, 3.+0.j])
```

```
[336]: data.real
```

```
[336]: array([1., 2., 3.])
```

```
[337]: data.imag
```

```
[337]: array([0., 0., 0.])
```

length of the array

```
[338]: len(np.arange(10,23))
```

```
[338]: 13
```

```
[339]: np.arange(10,23).size
```

```
[339]: 13
```

1.0.5 Difference between Python and NumPy Data Structure

```
[340]: mytuple = (1,2,3,4,5,6,7,8,9,10)
      # np.array(mytuple)
      mytuple * 3
```

```
[340]: (1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      10,
      1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      10,
      1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      10)
```

```
[341]: mytuple = (1,2,3,4,5,6,7,8,9,10)
      np.array(mytuple)
      np.array(mytuple) * 3
```

```
[341]: array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30])
```

1.0.6 NumPy shape

```
[342]: np1.shape #showing the shape of array, dimention and/or numer of values in  
       ↪array(in the current case as it is just one dimentional array)
```

```
[342]: (10,)
```

```
[343]: np2 = np.array([[1,2,3,4,5],[6,7,8,9,10]])  
       np2.shape
```

```
[343]: (2, 5)
```

```
[344]: np2
```

```
[344]: array([[ 1,  2,  3,  4,  5],  
             [ 6,  7,  8,  9, 10]])
```

```
[345]: np2.ravel() #returns the array, flattened
```

```
[345]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
[346]: np2 = np.arange(10) #creating NumPy array with arange() with 10 values  
       np2
```

```
[346]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[347]: np3 = np.arange(0,10,2) #usin arange steps option with value 2  
       np3
```

```
[347]: array([0, 2, 4, 6, 8])
```

```
[348]: np5 = np.zeros((2,10)) # creatin NumPy array with 2 dimentional with 10 items  
       ↪in each dimention  
       np5
```

```
[348]: array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
             [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
[349]: np6 = np.full((10), 6) #NumPy full() function to create NumPy single  
       ↪dimentional array with 10 values and all values are 6  
       np6
```

```
[349]: array([6, 6, 6, 6, 6, 6, 6, 6, 6, 6])
```

```
[350]: np7 = np.full((2, 10), 6) #NumPy full() function to create NumPy two  
       ↪dimentional array with 10 values and all values are 6  
       np7
```



```
[350]: array([[6, 6, 6, 6, 6, 6, 6, 6, 6, 6],
              [6, 6, 6, 6, 6, 6, 6, 6, 6, 6]])
```

```
[351]: a = np.empty([2, 3, 2])
a
```

```
[351]: array([[[[1.56690889e-316, 0.00000000e+000],
                [1.03977794e-312, 9.54898106e-313],
                [1.01855798e-312, 1.03977794e-312]],
               [[1.23075756e-312, 1.14587773e-312],
                [1.10343781e-312, 9.76118064e-313],
                [1.08221785e-312, 1.90979621e-312]]]])
```

```
[352]: #converting list to NumPy array
mylist = [1,2,3,4,5]
np8 = np.array(mylist)
np8
```

```
[352]: array([1, 2, 3, 4, 5])
```

```
[353]: #accessing ay specific item of NumPy array
np8[0]
```

```
[353]: 1
```

```
[354]: np8[2]
```

```
[354]: 3
```

```
[355]: #checkin the datatype of NumPy array
np8.dtype
```

```
[355]: dtype('int64')
```

1.0.7 Two dimentional arrays

```
[356]: my_array = np.arange(35)
my_array.shape = (7,5)
my_array
```

```
[356]: array([[ 0,  1,  2,  3,  4],
              [ 5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14],
              [15, 16, 17, 18, 19],
              [20, 21, 22, 23, 24],
              [25, 26, 27, 28, 29],
              [30, 31, 32, 33, 34]])
```

```
[30, 31, 32, 33, 34]))
```

```
[357]: my_array[2]
```

```
[357]: array([10, 11, 12, 13, 14])
```

```
[358]: my_array[-2]
```

```
[358]: array([25, 26, 27, 28, 29])
```

```
[359]: my_array[6,2]
```

```
[359]: 32
```

```
[360]: # my_array[6][21]
```

1.0.8 Three Dimentional Array

```
[361]: my3darray = np.arange(70)  
my3darray
```

```
[361]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
            17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,  
            34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,  
            51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,  
            68, 69])
```

```
[362]: my3darray.shape=(2,7,5)  
my3darray
```

```
[362]: array([[[ 0,  1,  2,  3,  4],  
              [ 5,  6,  7,  8,  9],  
              [10, 11, 12, 13, 14],  
              [15, 16, 17, 18, 19],  
              [20, 21, 22, 23, 24],  
              [25, 26, 27, 28, 29],  
              [30, 31, 32, 33, 34]],  
            [[35, 36, 37, 38, 39],  
              [40, 41, 42, 43, 44],  
              [45, 46, 47, 48, 49],  
              [50, 51, 52, 53, 54],  
              [55, 56, 57, 58, 59],  
              [60, 61, 62, 63, 64],  
              [65, 66, 67, 68, 69]]])
```

1.0.9 Basic Indexing and Slicing

```
[363]: np1 = np.array([1,2,3,4,5,6,7,8,9])
```

```
[364]: np1[1:5]
```

```
[364]: array([2, 3, 4, 5])
```

```
[365]: np1[1:]
```

```
[365]: array([2, 3, 4, 5, 6, 7, 8, 9])
```

```
[366]: np1[-3:-1]
```

```
[366]: array([7, 8])
```

```
[367]: np1[1:5:2] #with stepping 2
```

```
[367]: array([2, 4])
```

```
[368]: np1[::2] #prints start to end with stepping 2
```

```
[368]: array([1, 3, 5, 7, 9])
```

```
[369]: np1[::3] #prints start to end with stepping 3
```

```
[369]: array([1, 4, 7])
```

```
[370]: arr = np.arange(10)
      arr
```

```
[370]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[371]: arr[5]
```

```
[371]: 5
```

```
[372]: arr[5:8]
```

```
[372]: array([5, 6, 7])
```

```
[373]: arr[5:8] = 12
```

```
[374]: arr
```

```
[374]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

```
[375]: arr_slice = arr[5:8]
arr_slice
```

```
[375]: array([12, 12, 12])
```

```
[376]: arr_slice[1] = 12345
```

```
[377]: arr
```

```
[377]: array([ 0,  1,  2,  3,  4, 12, 12345, 12,  8,
              9])
```

The “bare” slice `[:]` will assign to all values in an array:

```
[378]: arr_slice[:] = 64
```

```
[379]: arr
```

```
[379]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
[380]: #slicing multi dimensional array
np2 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
np2[0:1,1:3]
```

```
[380]: array([[2, 3]])
```

```
[381]: np2[0:2,1:3]
```

```
[381]: array([[2, 3],
              [7, 8]])
```

```
[382]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

1.1 Copy Vs View for NumPy

```
[383]: import numpy as np
np1 = np.array([0,1,2,3,4,5])
```

```
[384]: #creating view
np2= np1.view()
np1 #original NP1
```

```
[384]: array([0, 1, 2, 3, 4, 5])
```

```
[385]: np2 #Original NP2
```

```
[385]: array([0, 1, 2, 3, 4, 5])
```

```
[386]: np1[0] = 41
```

```
[387]: np1 #original NP1
```

```
[387]: array([41, 1, 2, 3, 4, 5])
```

```
[388]: np2 #Original NP2
```

```
[388]: array([41, 1, 2, 3, 4, 5])
```

notice that the original and view both changed. It means when we change the original, the view also changes

Another example

```
[389]: rng = np.random.default_rng(seed=1701) # seed for reproducibility  
x1 = rng.integers(10, size=6) # one-dimensional array  
x1
```

```
[389]: array([9, 4, 0, 3, 8, 6])
```

```
[390]: x2 = rng.integers(10, size=(3, 4)) # two-dimensional array  
x2
```

```
[390]: array([[3, 1, 3, 7],  
            [4, 0, 2, 3],  
            [0, 0, 6, 9]])
```

```
[391]: x3 = rng.integers(10, size=(3, 4, 5)) # three-dimensional array  
x3
```

```
[391]: array([[[4, 3, 5, 5, 0],  
             [8, 3, 5, 2, 2],  
             [1, 8, 8, 5, 3],  
             [0, 0, 8, 5, 8]],  
            [[5, 1, 6, 2, 3],  
             [1, 2, 5, 6, 2],  
             [5, 2, 7, 9, 3],  
             [5, 6, 0, 2, 0]],  
            [[2, 9, 4, 3, 9],  
             [9, 2, 2, 4, 0],  
             [0, 3, 0, 0, 2]]])
```

```
[3, 2, 7, 4, 7]])
```

```
[392]: x2_sub = x2[:2, :2]
x2_sub
```

```
[392]: array([[3, 1],
           [4, 0]])
```

```
[393]: #copy function
np1 = np.array([0,1,2,3,4,5])
```

Now if we modify this subarray, we'll see that the original array is changed!

```
[394]: x2_sub[0, 0] = 99
x2_sub
```

```
[394]: array([[99, 1],
           [ 4, 0]])
```

```
[395]: x2
```

```
[395]: array([[99, 1, 3, 7],
           [ 4, 0, 2, 3],
           [ 0, 0, 6, 9]])
```

1.1.1 Creating a Copy

```
[396]: np1 = np.array([0,1,2,3,4,5])
np2 = np1.copy()
```

```
[397]: np1 #original NP1
```

```
[397]: array([0, 1, 2, 3, 4, 5])
```

```
[398]: np2 #original NP1
```

```
[398]: array([0, 1, 2, 3, 4, 5])
```

```
[399]: np1[0] = 41
```

```
[400]: np1 #original NP1
```

```
[400]: array([41, 1, 2, 3, 4, 5])
```

```
[401]: np2 #original NP1
```

```
[401]: array([0, 1, 2, 3, 4, 5])
```

notice that when we change the original but the copy has not changed. It simply indicates the copy is a separate copy of the original and has no links with the original

```
[402]: # now try to change the copy version to check if the original changes
```

```
[403]: np1 = np.array([0,1,2,3,4,5])
```

```
[404]: np2 = np1.copy()
```

```
[405]: np2
```

```
[405]: array([0, 1, 2, 3, 4, 5])
```

```
[406]: np1 #original NP1
```

```
[406]: array([0, 1, 2, 3, 4, 5])
```

```
[407]: np2[0] = 41
```

```
[408]: np1 #original NP1
```

```
[408]: array([0, 1, 2, 3, 4, 5])
```

```
[409]: np2 #original NP1
```

```
[409]: array([41, 1, 2, 3, 4, 5])
```

since copy is completely different array it only changes and it does not reflect the original array

1.2 Another example

```
[410]: x2 = rng.integers(10, size=(3, 4)) # two-dimensional array  
x2_sub_copy = x2[:2, :2].copy()  
x2_sub_copy
```

```
[410]: array([[9, 3],  
            [5, 0]])
```

If we now modify this subarray, the original array is not touched

```
[411]: x2_sub_copy[0, 0] = 42
```

```
[412]: x2_sub_copy
```

```
[412]: array([[42, 3],  
            [ 5, 0]])
```

```
[413]: x2
```

```
[413]: array([[9, 3, 6, 6],
             [5, 0, 2, 5],
             [6, 4, 8, 2]])
```

1.2.1 Indexing with Slices

```
[414]: arr
```

```
[414]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
[415]: arr[1:6]
```

```
[415]: array([ 1,  2,  3,  4, 64])
```

```
[416]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
[417]: arr2d
```

```
[417]: array([[1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]])
```

```
[418]: arr2d[:2]
```

```
[418]: array([[1, 2, 3],
             [4, 5, 6]])
```

```
[419]: arr2d[:2, 1:]
```

```
[419]: array([[2, 3],
             [5, 6]])
```

```
[420]: lower_dim_slice = arr2d[1, :2]
```

```
[421]: lower_dim_slice
```

```
[421]: array([4, 5])
```

```
[422]: lower_dim_slice.shape
```

```
[422]: (2,)
```

```
[423]: arr2d[:2, 2]
```

```
[423]: array([3, 6])
```

```
[424]: arr2d[:, :1]
```



```
[424]: array([[1],
             [4],
             [7]])
```

```
[425]: arr2d[:2, 1:] = 0
```

```
[426]: arr2d
```

```
[426]: array([[1, 0, 0],
             [4, 0, 0],
             [7, 8, 9]])
```

```
[427]: arr2d[1, :2]
```

```
[427]: array([4, 0])
```

```
[428]: my_vector = np.array([-15,-4,0,3,21,37,105])
my_vector
```

```
[428]: array([-15,  -4,   0,   3,  21,  37, 105])
```

```
[429]: my_vector[0]
```

```
[429]: -15
```

```
[430]: my_vector[1]
```

```
[430]: -4
```

```
[431]: # my_vector[305]
```

```
[432]: 305 % 7
```

```
[432]: 4
```

```
[433]: my_vector[305 % 7]
```

```
[433]: 21
```

check how it gives the 4th element of the my_vector array above

```
[434]: my_vector.size
```

```
[434]: 7
```

Boolean Indexing

```
[435]: names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
```

```
[436]: names
```

```
[436]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
[437]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, -4], [3, 4]])
```

```
[438]: data
```

```
[438]: array([[ 4,  7],
             [ 0,  2],
             [-5,  6],
             [ 0,  0],
             [ 1,  2],
             [-12, -4],
             [ 3,  4]])
```

```
[439]: names == "Bob"
```

```
[439]: array([ True, False, False,  True, False, False, False])
```

```
[440]: data[names == "Bob"]
```

```
[440]: array([[4, 7],
             [0, 0]])
```

```
[441]: data[names == "Bob", 1:]
```

```
[441]: array([[7],
             [0]])
```

```
[442]: data[names == "Bob", 1]
```

```
[442]: array([7, 0])
```

```
[443]: names != "Bob"
```

```
[443]: array([False,  True,  True, False,  True,  True,  True])
```

```
[444]: ~(names == "Bob")
```

```
[444]: array([False,  True,  True, False,  True,  True,  True])
```

```
[445]: data[~(names == "Bob")]
```

```
[445]: array([[ 0,  2],
             [-5,  6],
             [ 1,  2],
```

```
[-12, -4],  
[ 3, 4]])
```

```
[446]: cond = names == "Bob"
```

```
[447]: data[~cond]
```

```
[447]: array([[ 0,  2],  
            [-5,  6],  
            [ 1,  2],  
            [-12, -4],  
            [ 3,  4]])
```

```
[448]: mask = (names == "Bob") | (names == "Will")
```

```
[449]: mask
```

```
[449]: array([ True, False,  True,  True,  True, False, False])
```

```
[450]: data[mask]
```

```
[450]: array([[ 4,  7],  
            [-5,  6],  
            [ 0,  0],  
            [ 1,  2]])
```

```
[451]: data[data < 0] = 0
```

```
[452]: data
```

```
[452]: array([[4, 7],  
            [0, 2],  
            [0, 6],  
            [0, 0],  
            [1, 2],  
            [0, 0],  
            [3, 4]])
```

```
[453]: data[names != "Joe"] = 7
```

```
[454]: data
```

```
[454]: array([[7, 7],  
            [0, 2],  
            [7, 7],  
            [7, 7],  
            [7, 7],  
            [7, 7],  
            [7, 7]])
```

```
[0, 0],  
[3, 4]])
```

1.2.2 Another example

```
[455]: my_vector = np.array([-17,-3,0,4,7,19,109])
```

```
[456]: my_vector
```

```
[456]: array([-17,  -3,   0,   4,   7,  19, 109])
```

```
[457]: zero_mod_7_mask = 0 == (my_vector % 7)  
zero_mod_7_mask
```

```
[457]: array([False, False,  True, False,  True, False, False])
```

```
[458]: sub_array = my_vector[zero_mod_7_mask]  
sub_array
```

```
[458]: array([0, 7])
```

```
[459]: sub_array[sub_array > 0]
```

```
[459]: array([7])
```

1.2.3 NumPy logical Operator

Fancy Indexing

```
[460]: arr = np.zeros((8, 4))
```

```
[461]: for i in range(8):  
        arr[i] = i
```

```
[462]: arr
```

```
[462]: array([[0., 0., 0., 0.],  
             [1., 1., 1., 1.],  
             [2., 2., 2., 2.],  
             [3., 3., 3., 3.],  
             [4., 4., 4., 4.],  
             [5., 5., 5., 5.],  
             [6., 6., 6., 6.],  
             [7., 7., 7., 7.]])
```

```
[463]: arr[[4, 3, 0, 6]]
```

```
[463]: array([[4., 4., 4., 4.],
             [3., 3., 3., 3.],
             [0., 0., 0., 0.],
             [6., 6., 6., 6.]])
```

```
[464]: arr[[-3, -5, -7]]
```

```
[464]: array([[5., 5., 5., 5.],
             [3., 3., 3., 3.],
             [1., 1., 1., 1.]])
```

```
[465]: arr = np.arange(32).reshape((8, 4))
```

```
[466]: arr
```

```
[466]: array([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11],
             [12, 13, 14, 15],
             [16, 17, 18, 19],
             [20, 21, 22, 23],
             [24, 25, 26, 27],
             [28, 29, 30, 31]])
```

```
[467]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
[467]: array([ 4, 23, 29, 10])
```

```
[468]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
[468]: array([[ 4,  7,  5,  6],
             [20, 23, 21, 22],
             [28, 31, 29, 30],
             [ 8, 11,  9, 10]])
```

```
[469]: arr.shape
```

```
[469]: (8, 4)
```

1.2.4 NumPy universal Functions

```
[470]: np1 = np.array([0,1,2,3,4,5,6,7,8,9])
       np1
```

```
[470]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[471]: #sqrt function
np.sqrt((np1))
```

```
[471]: array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,
        2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])
```

```
[472]: #absolute value function
np1= np.array([-3,-2-1,0,1,2,3,4,5,6,7,8,9])
np.absolute((np1))
```

```
[472]: array([3, 3, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[473]: #exponent fujnction
np.exp((np1))
```

```
[473]: array([4.97870684e-02, 4.97870684e-02, 1.00000000e+00, 2.71828183e+00,
        7.38905610e+00, 2.00855369e+01, 5.45981500e+01, 1.48413159e+02,
        4.03428793e+02, 1.09663316e+03, 2.98095799e+03, 8.10308393e+03])
```

```
[474]: #min and max function
np.max((np1))
```

```
[474]: 9
```

```
[475]: np.min((np1))
```

```
[475]: -3
```

```
[476]: #dot product calculation
l1 = [1,2,3]
l2 = [4,5,6]
a1 = np.array((l1))
a2 = np.array((l2))
print("dot product via list")
dot =0
for i in range(len(l1)):
    dot += l1[i] * l2[i]
print(dot)

print("dot poroduct via NumPy array")
dot = np.dot(a1,a2)
print(dot)
```

```
dot product via list
32
dot poroduct via NumPy array
32
```

```
[477]: #linspace function  
np.linspace(5,15,9)
```

```
[477]: array([ 5. ,  6.25,  7.5 ,  8.75, 10. , 11.25, 12.5 , 13.75, 15.  ])
```

```
[478]: mylinspace = np.linspace(5,15,9, retstep=True)  
mylinspace
```

```
[478]: (array([ 5. ,  6.25,  7.5 ,  8.75, 10. , 11.25, 12.5 , 13.75, 15.  ]), 1.25)
```

```
[479]: mylinspace[1]
```

```
[479]: 1.25
```

```
[480]: #zeros() function  
np.zeros(6)
```

```
[480]: array([0., 0., 0., 0., 0., 0.])
```

```
[481]: #one function  
np.ones(6)
```

```
[481]: array([1., 1., 1., 1., 1., 1.])
```

```
[482]: np.zeros((5,5))
```

```
[482]: array([[0., 0., 0., 0., 0.],  
         [0., 0., 0., 0., 0.],  
         [0., 0., 0., 0., 0.],  
         [0., 0., 0., 0., 0.],  
         [0., 0., 0., 0., 0.]])
```

```
[483]: np.zeros((5,4,3))
```

```
[483]: array([[[0., 0., 0.],  
            [0., 0., 0.],  
            [0., 0., 0.],  
            [0., 0., 0.]],  
          [[0., 0., 0.],  
            [0., 0., 0.],  
            [0., 0., 0.],  
            [0., 0., 0.]],  
          [[0., 0., 0.],  
            [0., 0., 0.],  
            [0., 0., 0.],  
            [0., 0., 0.]])
```

```

        [0., 0., 0.]],

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]],

[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]])

```

1.2.5 NumPy DataTypes

```
[484]: np.zeros(11, dtype='int64')
```

```
[484]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
[485]: np.zeros(11)
```

```
[485]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
[ ]:
```

Transposing Arrays and Swapping Axes

```
[486]: arr = np.arange(15).reshape((3, 5))
```

```
[487]: arr
```

```
[487]: array([[ 0,  1,  2,  3,  4],
              [ 5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14]])
```

```
[488]: arr.T #returnsthe array,transposed
```

```
[488]: array([[ 0,  5, 10],
              [ 1,  6, 11],
              [ 2,  7, 12],
              [ 3,  8, 13],
              [ 4,  9, 14]])
```

```
[489]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])
```

```
[490]: arr
```



```
[490]: array([[ 0,  1,  0],
              [ 1,  2, -2],
              [ 6,  3,  2],
              [-1,  0, -1],
              [ 1,  0,  1]])
```

```
[491]: np.dot(arr.T, arr)
```

```
[491]: array([[39, 20, 12],
              [20, 14,  2],
              [12,  2, 10]])
```

```
[492]: arr.T @ arr
```

```
[492]: array([[39, 20, 12],
              [20, 14,  2],
              [12,  2, 10]])
```

```
[493]: arr
```

```
[493]: array([[ 0,  1,  0],
              [ 1,  2, -2],
              [ 6,  3,  2],
              [-1,  0, -1],
              [ 1,  0,  1]])
```

```
[494]: arr.swapaxes(0, 1)
```

```
[494]: array([[ 0,  1,  6, -1,  1],
              [ 1,  2,  3,  0,  0],
              [ 0, -2,  2, -1,  1]])
```

1.2.6 Pseudorandom Number Generation

```
[495]: samples = np.random.standard_normal(size=(4, 4))
```

```
[496]: samples
```

```
[496]: array([[ 0.64388249, -0.83486572,  0.81875284, -0.96590295],
              [-2.19260694, -0.63743414,  0.07675669, -1.00403397],
              [ 0.22750889,  0.04179036, -0.52341692, -0.23373595],
              [-0.55381454, -0.60664394, -2.29954216,  0.56674033]])
```

```
[497]: from random import normalvariate
```

```
[498]: N = 1_000_000
```

1.2.7 NumPy Reshaping

```
[499]: np1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])  
np1
```

```
[499]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
[500]: np1.shape
```

```
[500]: (12,)
```

```
[501]: np2 = np1.reshape(3,4)  
np2
```

```
[501]: array([[ 1,  2,  3,  4],  
            [ 5,  6,  7,  8],  
            [ 9, 10, 11, 12]])
```

```
[502]: np2.shape
```

```
[502]: (3, 4)
```

```
[503]: np3 = np1.reshape(2,3,2)  
np3
```

```
[503]: array([[[ 1,  2],  
              [ 3,  4],  
              [ 5,  6]],  
            [[ 7,  8],  
              [ 9, 10],  
              [11, 12]]])
```

```
[504]: np3.shape
```

```
[504]: (2, 3, 2)
```

```
[505]: np4 = np3.reshape(-1)  
np4
```

```
[505]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
[506]: np4.shape
```

```
[506]: (12,)
```

1.2.8 Another example

```
[507]: grid = np.arange(1, 10).reshape(3, 3)
       grid
```

```
[507]: array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

Note that for this to work, the size of the initial array must match the size of the reshaped array, and in most cases the reshape method will return a no-copy view of the initial array.

```
[508]: x = np.array([1,2,3])
       x.reshape((1,3))
```

```
[508]: array([[1, 2, 3]])
```

```
[509]: x.reshape((3,1))
```

```
[509]: array([[1],
              [2],
              [3]])
```

```
[510]: x[np.newaxis, :]
```

```
[510]: array([[1, 2, 3]])
```

```
[511]: x[:, np.newaxis]
```

```
[511]: array([[1],
              [2],
              [3]])
```

1.2.9 Stacking together different arrays

```
[ ]:
```

1.2.10 Iterate through NumPy array

```
[512]: np1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
```

```
[513]: np1
```

```
[513]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
np2 = np.array([[1,2,3,4,5],[6,7,8,9,10]]) np2
```

```
[514]: for x in np.nditer(np2):  
        print(x)
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

1.2.11 Concatenation of Arrays

```
[515]: x = np.array([1,2,3])  
        y = np.array([3,2,1])  
        np.concatenate([x,y])
```

```
[515]: array([1, 2, 3, 3, 2, 1])
```

```
[516]: z = np.array([3,5,6,8,67])  
        np.concatenate([x,y,z])
```

```
[516]: array([ 1,  2,  3,  3,  2,  1,  3,  5,  6,  8, 67])
```

```
[517]: grid = np.array([[1, 2, 3],[4, 5, 6]])  
        np.concatenate([grid,grid])
```

```
[517]: array([[1, 2, 3],  
             [4, 5, 6],  
             [1, 2, 3],  
             [4, 5, 6]])
```

```
[518]: # concatenate along the second axis (zero-indexed)  
        np.concatenate([grid, grid], axis=1)
```

```
[518]: array([[1, 2, 3, 1, 2, 3],  
             [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions

```
[519]: # vertically stack the arrays  
        np.vstack([x, grid])
```

```
[519]: array([[1, 2, 3],
           [1, 2, 3],
           [4, 5, 6]])
```

```
[520]: # horizontally stack the arrays
       y = np.array([[99],[99]])
       np.hstack([grid, y])
```

```
[520]: array([[ 1,  2,  3, 99],
              [ 4,  5,  6, 99]])
```

1.2.12 Splitting of Arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points

```
[521]: x = [1, 2, 3, 99, 99, 3, 2, 1]
       x1, x2, x3 = np.split(x, [3,5])
```

```
[522]: x1
```

```
[522]: array([1, 2, 3])
```

```
[523]: x2
```

```
[523]: array([99, 99])
```

```
[524]: x3
```

```
[524]: array([3, 2, 1])
```

1.3 Sorting NumPy array

```
[525]: np1 = np.array([5,1,78,9,1,5786,2,4,7,0])
       np.sort(np1)
```

```
[525]: array([ 0,  1,  1,  2,  4,  5,  7,  9, 78, 5786])
```

```
[526]: np2 = np.array(['Zakir', 'John', 'Tina', 'Aaron'])
       np.sort(np2)
```

```
[526]: array(['Aaron', 'John', 'Tina', 'Zakir'], dtype='<U5')
```

```
[527]: np3 = np.array([True, False, True, False, True, True, False])
       np.sort(np3)
```

```
[527]: array([False, False, False,  True,  True,  True,  True])
```

```
[528]: np4 = np.array([[3,7,1,5,89],[3,8,90,2,90]])
      np.sort(np4)
```

```
[528]: array([[ 1,  3,  5,  7, 89],
      [ 2,  3,  8, 90, 90]])
```

```
[529]: grid = np.arange(16).reshape((4, 4))
      grid
```

```
[529]: array([[ 0,  1,  2,  3],
      [ 4,  5,  6,  7],
      [ 8,  9, 10, 11],
      [12, 13, 14, 15]])
```

```
[530]: upper, lower = np.vsplit(grid, [2])
      print(upper)
      print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
[531]: left, right = np.hsplit(grid, [2])
      print(left)
      print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

1.3.1 NumPy Searching

```
[532]: np1 = np.array([1,2,3,4,5,3,6,68678,6,78,87,78])
      print(np1)
      print(np.where(np1==3))
```

```
[ 1  2  3  4  5  3  6 68678  6  78  87  78]
(array([2, 5]),)
```

```
[533]: a = np.where(np1==3)[0]
      print(a)
```

[2 5]

```
[534]: print(np1[a[0]])
```

3

```
[535]: ## prints even numbers
y = np.where(np1 % 2 == 0)
print(np1)
print(y)
print(y[0])
```

```
[ 1  2  3  4  5  3  6 68678  6  78  87  78]
(array([ 1,  3,  6,  7,  8,  9, 11]),)
[ 1  3  6  7  8  9 11]
```

```
[536]: ## prints odd numbers
y = np.where(np1 % 2 == 1)
print(np1)
print(y)
print(y[0])
```

```
[ 1  2  3  4  5  3  6 68678  6  78  87  78]
(array([ 0,  2,  4,  5, 10]),)
[ 0  2  4  5 10]
```

Filtering NumPy Arrays with Boolean Index lists

```
[537]: np1 = np.array([1,2,3,4,5,6,7,8,9,10])
x = [True, True, False, False, True, False, False, False, False, False]
```

```
[538]: print(np1)
print(np1[x])
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[1 2 5]
```

```
[539]: filtered = []
for thing in np1:
    if thing % 2 == 0:
        filtered.append(True)
    else:
        filtered.append(False)
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[False, True, False, True, False, True, False, True, False, True]
[ 2  4  6  8 10]
```

```
[540]: filtered = []
for thing in np1:
    if thing > 5:
        filtered.append(True)
    else:
        filtered.append(False)
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[False, False, False, False, False, True, True, True, True, True]
[ 6  7  8  9 10]
```

```
[541]: filtered = np1 % 2 == 0
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[False  True False  True False  True False  True False  True]
[ 2  4  6  8 10]
```

```
[542]: filtered = np1 > 5
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[False False False False False  True  True  True  True  True]
[ 6  7  8  9 10]
```

```
[ ]:
```